**Australian Government**

**Department of Defence**

Defence Science and Technology Organisation

# Solving multi-dimensional problems of gas dynamics using MATLAB®

***L. K. Antanovskii***

**Weapons Systems Division**

**Defence Science and Technology Organisation**

**DSTO–TR–2139**

## ABSTRACT

This report describes an implementation of a Godunov-type solver for gas dynamics equations in MATLAB®. The main attention is paid to providing a generic code that can be easily adapted to particular problems in one, two or three dimensions. This is achieved by employing a cell connectivity matrix thus allowing one to use various structured and unstructured meshes without modification of the core solver. The code has been thoroughly tested for MATLAB Version 7.6 (Release 2008a).

*Published by*

*DSTO Defence Science and Technology Organisation*
*PO Box 1500*
*Edinburgh, South Australia 5111, Australia*

*Telephone: (08) 8259 5555*
*Facsimile: (08) 8259 6567*

*AR No. 014-204*
*June, 2008*

## Solving multi-dimensional problems of gas dynamics using MATLAB®

## Executive Summary

In many circumstances it is required to simulate blast propagation in complex three-dimensional domains in order to estimate pressure and temperature fields at some distance from the source of explosion. Though this is the classical problem of gas dynamics whose solution is implemented in many commercial software packages, the use of such packages is sometimes limited to a particular platform, specific mesh type and proprietary input/output file formats, and requires a long learning curve associated with input data preparation (pre-processing) and visualisation of the obtained results (post-processing). At the same time commercial software may not be open to model extension and may not be easily embedded into other systems.

This report describes an implementation of a Godunov-type solver for gas dynamics equations in MATLAB®. The main attention is paid to providing a generic code that can be easily adapted to particular problems in one, two or three dimensions. This is achieved by employing a cell connectivity matrix thus allowing one to use various structured and unstructured meshes without modification of the core solver. The code has been thoroughly tested for MATLAB Version 7.6 (Release 2008a).

The work has been performed within the "Blast Modelling for Cordon Assessment and Bomb Scene Examination" research project NS 07/002.

# Author

**Leonid Antanovskii**
*Weapons Systems Division*

Leonid Antanovskii obtained an MSc Degree with Distinction in Mechanics and Applied Mathematics from the Novosibirsk State University (Russia) in 1979 and a PhD in Mechanics of Fluid, Gas and Plasma from the Lavrentyev Institute of Hydrodynamics (Russian Academy of Science) in 1982. Since graduation he worked for the Lavrentyev Institute of Hydrodynamics in the area of fluid mechanics. In 1991–1994 he worked for Microgravity Advanced Research & Support Center (Italy) under the auspices of the European Space Agency, being involved in the modelling of complex physical phenomena predominantly arising in low-gravity environment. In 1994–1995 he worked at the University of the West Indies (Trinidad & Tobago), in 1996–1998 in Moldflow (Australia), and in 1998–2000 in Advanced CAE Technologies (USA). In 2000–2007 he worked in private industry in Australia.

Leonid Antanovskii joined DSTO in February 2007 where his current research interests include development of vulnerability and lethality models for weapon–target interaction.

# Contents

**1 Introduction** **1**

**2 Elements of gas dynamics** **1**

**3 The Riemann problem** **3**

**4 Numerical model** **6**

**5 Gas dynamics toolbox** **8**

**6 Usage examples** **16**

6.1 Sod's shock tube . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

6.2 Rotationally symmetric blast . . . . . . . . . . . . . . . . . . . . . . . . . . 18

6.3 Two-dimensional blast propagation . . . . . . . . . . . . . . . . . . . . . . . 21

6.4 Three-dimensional blast propagation . . . . . . . . . . . . . . . . . . . . . . 26

**7 Conclusion** **32**

**References** **32**

# Figures

1 Dimensionless Hugoniot–Poisson curves. . . . . . . . . . . . . . . . . . . . . . . . 5
2 Gas state in Sod's shock tube. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3 Blast waves from a spherical charge. . . . . . . . . . . . . . . . . . . . . . . . . 19
4 Two-dimensional blast propagation; pressure field surfaces. . . . . . . . . . . 24
5 Two-dimensional blast propagation; temperature field surfaces. . . . . . . . . 25
6 Three-dimensional blast propagation; pressure field slices. . . . . . . . . . . . 30
7 Three-dimensional blast propagation; temperature field slices. . . . . . . . . . 31

# 1 Introduction

In many circumstances it is required to simulate blast propagation in complex three-dimensional domains in order to estimate pressure and temperature fields at some distance from the source of explosion. Though this is the classical problem of gas dynamics whose solution is implemented in many commercial software packages, the use of such packages is sometimes limited to a particular platform, specific mesh type and proprietary input/output file formats, and requires a long learning curve associated with input data preparation (pre-processing) and visualisation of the obtained results (post-processing). At the same time commercial software may not be open to model extension and may not be easily embedded into other systems.

MATLAB® provides a nice environment for numerical simulation of real-world problems with integrated visualisation, powerful scripting framework, and fast algorithms implemented. Many books are devoted to the application of MATLAB to various problems of science and engineering (see e.g. [Danaila et al. 2007, Klee 2007, Gilat 2008]).

In this report, two compact MATLAB files (M-files) constituting the development of a Gas Dynamics Toolbox (MATLAB library) are presented which are sufficient to run from a user-defined script and simulate gas dynamics problems. A robust Godunov-type solver [Godunov 1959, Richtmyer & Morton 1967, Holt 1977, Toro 1999] based on the exact Riemann solver for flux calculation between control cells is employed. Though there exist many advanced schemes for simulation of gas dynamics equations with higher accuracy (see e.g. [van Leer 1979]), the first-order accurate Godunov solver provides a highly desirable monotone numerical scheme and is easily extendable to multi-dimensional problems. The history of discovery of this algorithm based on deep insight into the physics of shock and rarefaction waves is given in [Godunov 1999].

# 2 Elements of gas dynamics

Let $\omega$ be a control volume with piecewise smooth boundary $\partial\omega$ oriented by inward normal unit vector $\mathbf{n}$. The classical conservation laws of mass, momentum and energy for gas flow, written for arbitrary $\omega$, are as follows

$$\frac{\mathrm{d}}{\mathrm{d}t}\int\limits_{\omega} \rho \,\mathrm{d}V = \int\limits_{\partial\omega} \rho\,\mathbf{v}\cdot\mathbf{n}\,\mathrm{d}A\,, \tag{1}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\int\limits_{\omega} \rho\,\mathbf{v}\,\mathrm{d}V = \int\limits_{\partial\omega} (\rho\,\mathbf{v}\,\mathbf{v}\cdot\mathbf{n} + p\,\mathbf{n})\;\mathrm{d}A\,, \tag{2}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\int\limits_{\omega} \rho\left(e+\frac{1}{2}\,|\mathbf{v}|^2\right)\,\mathrm{d}V = \int\limits_{\partial\omega} \left[\rho\left(e+\frac{1}{2}\,|\mathbf{v}|^2\right)+p\right]\,\mathbf{v}\cdot\mathbf{n}\,\mathrm{d}A\,. \tag{3}$$

Here $t$ is time, $\rho$ density, $\mathbf{v}$ velocity, $p$ pressure, and $e$ specific internal energy. Symbols $\mathrm{d}V$ and $\mathrm{d}A$ denote volume and area elements, respectively. The effect of gravity is ignored.

Since the control volume $\omega$ is arbitrary, the integral conservation laws (1)–(3) imply the following differential conservation equations

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho\, \mathbf{v}) = 0\,, \tag{4}$$

$$\frac{\partial (\rho\, \mathbf{v})}{\partial t} + \nabla \cdot (\rho\, \mathbf{v} \otimes \mathbf{v} + p\, G) = \mathbf{0}\,, \tag{5}$$

$$\frac{\partial}{\partial t}\left[\rho\left(e + \frac{1}{2}|\mathbf{v}|^2\right)\right] + \nabla \cdot \left\{\left[\rho\left(e + \frac{1}{2}|\mathbf{v}|^2\right) + p\right]\mathbf{v}\right\} = 0\,, \tag{6}$$

where $\nabla$ denotes the gradient operator, $\otimes$ the tensor product, and $G$ the metric tensor. The conservative form of gas dynamics equations admits discontinuous solutions such as shock waves and contact discontinuities. In this case temporal and spatial derivatives have to be considered in the generalised sense. For continuous solutions such as rarefaction waves the conservative Euler equations (4)–(6) can be further simplified to

$$\frac{\partial \rho}{\partial t} + \mathbf{v} \cdot \nabla \rho + \rho\, \nabla \cdot \mathbf{v} = 0\,, \tag{7}$$

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}\right) + \nabla p = \mathbf{0}\,, \tag{8}$$

$$\rho\left(\frac{\partial e}{\partial t} + \mathbf{v} \cdot \nabla e\right) + p\, \nabla \cdot \mathbf{v} = 0\,. \tag{9}$$

As a consequence of equations (7), (9) and the second law of thermodynamics

$$\theta\, \mathrm{d}s = \mathrm{d}e + p\, \mathrm{d}\left(\frac{1}{\rho}\right) \tag{10}$$

where $\theta$ and $s$ are the absolute temperature and specific entropy, respectively, the substantial derivative of entropy vanishes:

$$\frac{\mathrm{d}s}{\mathrm{d}t} \equiv \frac{\partial s}{\partial t} + \mathbf{v} \cdot \nabla s = 0\,. \tag{11}$$

In other words, entropy at a particle remains constant in a continuous gas flow. Note that entropy at a particle undergoing a shock wave always increases.

The fundamental thermodynamic relations for ideal polytropic gas result in the following state equation [Courant & Friedrichs 1948]

$$p = (\gamma - 1)\, \rho\, e \tag{12}$$

where $\gamma$ is the ratio of the specific heat at constant pressure to the specific heat at constant volume. Indeed, by the definition, gas is ideal (thermally perfect) if its thermodynamic state satisfies Clapeyron's equation

$$p = R\, \rho\, \theta \tag{13}$$

where $R$ is the specific gas constant defined as the ratio of the universal gas constant to the molar mass of gas (or, equivalently, the ratio of the Boltzmann constant to the average mass of the gas molecule). From the second law of thermodynamics (10) it is straightforward to deduce that the specific internal energy of ideal gas is a function of the absolute temperature alone. Ideal gas is polytropic (calorically perfect) if $e$ is a linear function of $\theta$, namely

$$e = c_v\,\theta \tag{14}$$

where $c_v$ is the specific heat at constant volume. In this case the specific enthalpy $h$, defined by the expression $h = e + p/\rho$, is also a linear function of the absolute temperature

$$h = c_p\,\theta \tag{15}$$

where $c_p = c_v + R$ is the specific heat at constant pressure. These formulae result in expression (12) with $\gamma = c_p/c_v$. Note that the sound speed $a$ of a polytropic gas is given by the formula

$$a = \sqrt{\frac{\gamma\,p}{\rho}}\,. \tag{16}$$

Let us introduce the volumetric density of momentum, $\mathbf{u} = \rho\,\mathbf{v}$, and the volumetric density of total energy, $\varepsilon = \rho\left(e + \frac{1}{2}\,|\mathbf{v}|^2\right)$. In terms of these conserved variables the conservation laws (1)–(3) take the succinct form

$$\frac{\mathrm{d}}{\mathrm{d}t}\int\limits_{\omega}[\rho,\,\mathbf{u},\,\varepsilon]\;\mathrm{d}V = \int\limits_{\partial\omega}\left[\mathbf{u}\cdot\mathbf{n},\;\frac{1}{\rho}\,\mathbf{u}\,\mathbf{u}\cdot\mathbf{n} + p\,\mathbf{n},\;\frac{\varepsilon + p}{\rho}\,\mathbf{u}\cdot\mathbf{n}\right]\mathrm{d}A \tag{17}$$

where

$$p = (\gamma - 1)\left(\varepsilon - \frac{|\mathbf{u}|^2}{2\rho}\right) \tag{18}$$

according to the equation of state (12).

Most numerical algorithms, including Godunov-type solvers, are based on the above integral equations. The main problem is related to the correct determination of the fluxes of mass, momentum and energy through the interface separating two control volumes in terms of the cell values of the conserved variables.

## 3 The Riemann problem

The one-dimensional Riemann problem is important for understanding the physics of compression and expansion waves, and provides an exact expression for fluxes between two adjacent uniform states of gas separated by an infinite plane. The solution of the Riemann problem is described in many texts on gas dynamics (see e.g. [Courant & Friedrichs 1948]). In this section the solution is outlined without providing full details.

Consider the propagation of a shock wave or rarefaction fan into a uniform gas in the direction of Cartesian coordinate $x$ with front initially positioned at $x = 0$. It is assumed that the gas flow does not depend on the remaining Cartesian coordinates, and velocity field has the only nonzero component $v$ in the $x$-direction. Denote given pre-state of gas initially occupying the half-space $\{x > 0\}$ by $(\rho_o,\, p_o,\, v_o)$. Let us determine all the possible post-states $(\rho_*,\, p_*,\, v_*)$ of gas initially occupying the complementary half-space $\{x < 0\}$ and connected with the given pre-state by a shock wave or simple rarefaction fan, both propagating in the positive direction of $x$ relative to $v_o$. It turns out that the post-states cannot be arbitrary but form a one-dimensional manifold (curve). This curve can be parametrised by the post-state pressure $p_*$, and therefore the density $\rho_*$ and velocity $v_*$ are known functions of $p_*$ and the pre-state $(\rho_o,\, p_o,\, v_o)$. These functions are piecewise analytic, described by two different expressions for $p_* > p_o$ (Hugoniot curve) and $p_* < p_o$ (Poisson curve). The Hugoniot and Poisson curves join smoothly at $p_* = p_o$ up to the continuous second-order derivative and form the combined Hugoniot–Poisson adiabatic curve given by the formulae

$$\frac{\rho_*}{\rho_o} = R_\gamma\left(\frac{p_*}{p_o}\right) , \tag{19}$$

$$\frac{v_* - v_o}{a_o} = V_\gamma\left(\frac{p_*}{p_o}\right) . \tag{20}$$

Here

$$R_\gamma(q) = \begin{cases} \dfrac{q + \frac{\gamma-1}{\gamma+1}}{\frac{\gamma-1}{\gamma+1}\, q + 1} & (q \geq 1) \\[2ex] q^{1/\gamma} & (0 \leq q < 1) \end{cases} \tag{21}$$

$$V_\gamma(q) = \begin{cases} \dfrac{q - 1}{\sqrt{\gamma\left(\frac{\gamma+1}{2}\, q + \frac{\gamma-1}{2}\right)}} & (q \geq 1) \\[2ex] \dfrac{2}{\gamma - 1}\left(q^{\frac{\gamma-1}{2\gamma}} - 1\right) & (0 \leq q < 1) \end{cases} \tag{22}$$

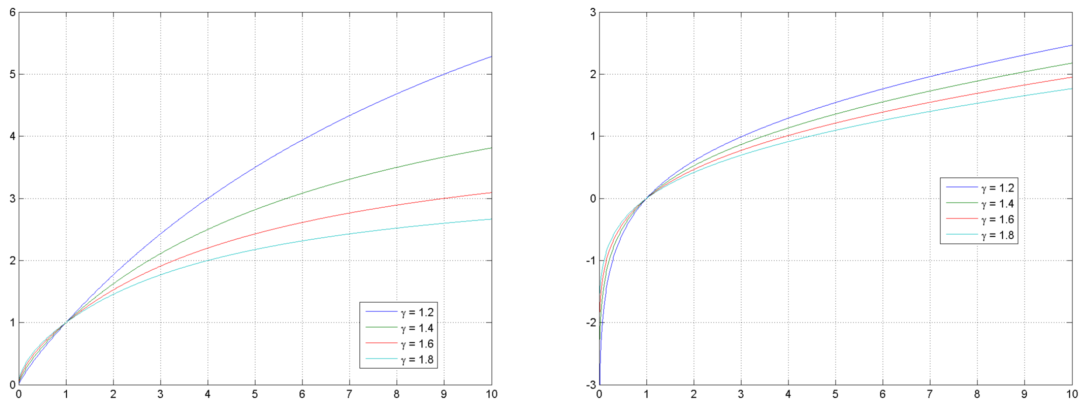
and $a_o$ is the pre-state sound speed of gas. The graphs of the functions $R_\gamma(q)$ and $V_\gamma(q)$ are displayed in Figure 1. In acoustic approximation of small $p_* - p_o$ (compared to $p_o$) equations (19) and (20) reduce to

$$\rho_* - \rho_o = \frac{p_* - p_o}{a_o^2} , \quad v_* - v_o = \frac{p_* - p_o}{\rho_o\, a_o} . \tag{23}$$

The product $\rho\, a$ is called impedance.

Note that the Hugoniot curve $(q > 1)$ is derived from the Rankine–Hugoniot conditions at the shock wave, whereas the Poisson curve $(0 \leq q < 1)$ is obtained from the exact solution of gas dynamics equations written in terms of Riemann invariants. The gas state is obtained as an explicit similarity solution (depending on $\xi = x/t$) to gas dynamics equations. For $p_* > p_o$ the compression Riemann wave is given by the piecewise constant profile of density, pressure, and velocity

$$(\rho,\, p,\, v) = \begin{cases} (\rho_o,\, p_o,\, v_o) & (v_s < \xi) \\ (\rho_*,\, p_*,\, v_*) & (\xi < v_s) \end{cases} \tag{24}$$



(i) Plots of $R_\gamma(q)$. (ii) Plots of $V_\gamma(q)$.

***Figure 1:*** *Dimensionless Hugoniot–Poisson curves for various $\gamma$.*

where

$$v_s = \frac{\rho_* \, v_* - \rho_o \, v_o}{\rho_* - \rho_o} \tag{25}$$

is the shock speed. For $0 \le p_* < p_o$ the expansion Riemann wave is described by the following continuous functions

$$(\rho,\, p,\, v) = \begin{cases} (\rho_o,\, p_o,\, v_o) & (v_o + a_o \le \xi) \\ (\hat\rho(\xi),\, \hat p(\xi),\, \hat v(\xi)) & (v_* + a_* < \xi < v_o + a_o) \\ (\rho_*,\, p_*,\, v_*) & (\xi \le v_* + a_*) \end{cases} \tag{26}$$

where $a_*$ is the post-state sound speed of gas, and

$$\hat\rho(\xi) = \rho_o \left[ \frac{\hat a(\xi)}{a_o} \right]^{\frac{2}{\gamma-1}} , \tag{27}$$

$$\hat p(\xi) = p_o \left[ \frac{\hat\rho(\xi)}{\rho_o} \right]^{\gamma} , \tag{28}$$

$$\hat v(\xi) = \xi - \hat a(\xi) \, , \tag{29}$$

$$\hat a(\xi) = a_o - \frac{\gamma - 1}{\gamma + 1} \, (v_o + a_o - \xi) \ . \tag{30}$$

It is straightforward to check that, since the state $(\rho_*,\, p_*,\, v_*)$ belongs to the Poisson curve, the gas properties in the expansion wave, given by (26), vary continuously.

Note that equation (19) expressed in terms of pressure $p$ as a function of specific volume $1/\rho$ is actually called the Hugoniot–Poisson curve. Conceptually, the latter variables are just specialised coordinates on the Hugoniot–Poisson manifold of all possible post-states.

Now consider two arbitrary uniform states of gas initially separated by some plane without assumption that the velocity vectors are perpendicular to the plane. To determine the breakdown of this configuration when time increases the above solutions can be used. Indeed, consider either of half-planes and make Galilean transformation to a new coordinate system in which the uniform velocity field is perpendicular to the plane. Then apply the above analytic formulae to describe all possible post-states parametrised by the post-state pressure. The two post-states defining two Riemann waves have to be connected to provide global solution. Since the gas density and tangential velocity can be discontinuous (due to potentially different Galilean transformations), the only kind of interface between the uniform post-states can be a contact discontinuity. Contact interfaces are defined by the conditions of continuity of pressure and the normal component of velocity. These conditions are readily achieved by the above explicit solutions. Indeed, the only condition to satisfy is the continuity of the normal component of the post-state velocity as a function of common post-state pressure $p_*$. This results in the following algebraic equation to solve

$$f\,(p_*) \equiv v_+ - v_- + a_+\,V_\gamma\left(\frac{p_*}{p_+}\right) + a_-\,V_\gamma\left(\frac{p_*}{p_-}\right) = 0 \tag{31}$$

where the subscripts $\pm$ supply the two sets of the pre-state variables of gas initially occupying half-spaces $x > 0$ and $x < 0$ respectively ($v_\pm$ are the normal components of the pre-state velocities). Since the combined Hugoniot–Poission adiabatic curve is globally smooth, the fast converging Newton–Raphson method can be applied to solve the non-linear equation (31). The acoustic approximation is frequently used as an initial guess in the iterative procedure. Since $f\,(p_*)$ increases monotonically and $f(+\infty) = +\infty$, a unique solution $p_*$ exists if and only if $f(0) \le 0$ which is equivalent to the condition

$$v_+ - v_- \le \frac{2\,(a_+ + a_-)}{\gamma - 1}\,. \tag{32}$$

Some care should be taken in the situation of strong rarefaction waves ($f(0) > 0$) leading to vacuum condition [Toro 1999].

An alternative Riemann solver can be obtained by inverting equation (20) and solving a non-linear equation for the post-state velocity $v_*$. The comparative performance analysis of various exact Riemann solvers is given in [Gottlieb & Groth 1988].

# 4 Numerical model

Let $\{\omega_i\}$ ($i = 1, \dots, N_c$) constitute tessellation of the computational domain $\mathcal{D}$ where $N_c$ is the total number of cells (control volumes). The state of the gas dynamics flow is approximated by the cell average values

$$S_i = \frac{1}{V_i}\int\limits_{\omega_i} [\rho,\, \mathbf{u},\, \varepsilon]\;\mathrm{d}V \tag{33}$$

and the fluxes by the surface integrals

$$R_i = \frac{1}{V_i}\int\limits_{\partial\omega_i} \left[\mathbf{u}\cdot\mathbf{n},\; \frac{1}{\rho}\,\mathbf{u}\,\mathbf{u}\cdot\mathbf{n} + p\,\mathbf{n},\; \frac{\varepsilon + p}{\rho}\,\mathbf{u}\cdot\mathbf{n}\right]\;\mathrm{d}A \tag{34}$$

where

$$V_i = \int\limits_{\omega_i} \mathrm{d}V \tag{35}$$

is the cell volume. According to the conservation laws (17) and the equation of state (18), the following system of ordinary differential equations results

$$\frac{\mathrm{d}S_i}{\mathrm{d}t} = R_i \, . \tag{36}$$

In order to complete this system of dynamic equations one needs to express the rates $R_i$ in terms of the states $S_i$. This is done by employing the Riemann solver which allows one to explicitly determine the fluxes between two cells in contact, provided that the gas states in the cells are approximated by constant values and the cells are separated by a planar patch. In this case the cell average values $S_i$ are approximated by the first order quadrature formulae

$$S_i = [\rho_i, \, \mathbf{u}_i, \, \varepsilon_i] \; . \tag{37}$$

Let us assume that each cell $\omega_i$ is a polyhedron. The collection $\{\omega_i\}$ $(i = 1, \ldots, N_c)$ can be potentially composed of polyhedra of different type, which do not necessarily share facets. Let $\{\varphi_k\}$ $(k = 1, \ldots, N_f)$ be the collection of all distinct atomic facets of the polyhedra where $N_f$ denotes the total number of facets. The term ‘atomic facet’ means the following. If two polyhedra in contact do not share a facet, only the smaller (atomic) facet is added to the collection of facets $\varphi_k$. For example, when two tetrahedra are attached to one face of a cube (regular hexahedron), only the two triangular faces of the tetrahedra covering the square face of the cube are accounted for. In this case the square face is the union of the two atomic triangular faces.

In order to uniformly treat boundary conditions, it is helpful to attach a ghost cell to each boundary facet. The state of the ghost cell must be specified by the user to model appropriate boundary conditions [Oran & Boris 1987]. This approach slightly increases the total number $N_c$ of cells but, as a compensation, all the facets become internal that greatly simplifies the logic of computation. It is worthwhile noting that periodic boundary conditions can be handled by the cell connectivity matrix alone without a need to introduce ghost cells.

Denote the area of facet $\varphi_k$ by $A_k$, select a unit normal vector $\mathbf{n}_k$, and introduce a $N_f$-by-2 facet-to-cell connectivity matrix $\alpha$ with entries defined as follows. If facet $\varphi_k$ belongs to the intersection $\partial\omega_{i_1} \cap \partial\omega_{i_2}$ and $\mathbf{n}_k$ points from $\omega_{i_1}$ to $\omega_{i_2}$, then set $\alpha(k, 1) = i_1$ and $\alpha(k, 2) = i_2$. Mathematically, the cell connectivity matrix $\alpha$ defines a directed graph $(\mathcal{V}, \mathcal{E})$ where vertices $\mathcal{V}$ are cells (including ghost cells) and edges $\mathcal{E}$ are facets. In particular, a basic theorem of graph theory [Diestel 2005] immediately provides the following relation

$$2\,|E| = \sum_{v \in V} \deg{(v)} \tag{38}$$

where $\deg{(v)}$ is the degree (or valency) of a vertex $v$ defined as the number of connected neighbours, and the standard symbol $|\mathcal{S}|$ denotes the cardinality of a set $\mathcal{S}$ (the number of elements). This relation can be used to relate the number $N_f$ of facets with the total number $N_c$ of cells (including ghost cells) and the number $N_g$ of ghost cells. For example, if each cell $\omega_i$ is a $n$-hedron, then

$$2N_f = n\ (N_c - N_g) + N_g < n\,N_c\,. \tag{39}$$

Indeed, the degree of each ordinary cell is equal to $n$ (the number of faces) and the degree of each ghost cell is equal to 1 by construction.

The numerical algorithm is straightforward. First update the states of the ghost cells according to the imposed boundary conditions. For example, for an ideally reflective wall or symmetry plane, set the ghost cell state equal to that of the neighbouring boundary cell except for the normal component of momentum which must have the opposite sign. For non-reflective (open-end) boundary condition, set the ghost cell state identical to the boundary cell state. Then for given $S_i$ at some instant $t$, calculate the cell values of velocity $\mathbf{v}_i = \mathbf{u}_i/\rho_i$ and pressure $p_i$ from formula (18). Zero out the array of rates $R_i$. For each facet $\varphi_k$ solve the Riemann problem using the given gas states in cells $\omega_{i_1}$ and $\omega_{i_2}$ where $i_1 = \alpha(k,1)$ and $i_2 = \alpha(k,2)$. Knowing the gas state at the facet calculate the fluxes of mass, momentum and energy along the unit normal vector $\mathbf{n}_k$, multiply by the facet area $A_k$, and add to the rate $R_{i_2}$ but subtract from the rate $R_{i_1}$ according to the choice of the normal $\mathbf{n}_k$. Finally, divide the calculated rates $R_i$ by volumes $V_i$.

This procedure defines the cell rates $\{R_i\}$ as a function of the cell states $\{S_i\}$ and therefore the evolution of the gas flow can be calculated by the explicit Euler scheme. There is not much reason to use the higher-order Runge–Kutta formulae as the similarity solution to the Riemann problem provides time-independent fluxes at facets. The size of the time step must be selected according to the Courant–Friedrichs–Lewy (CFL) condition which requires estimation of velocity and sound speed at the cell interfaces. Physically, this condition limits the time step to the value small enough not to allow Riemann waves to propagate more than half size of a control cell. Note that the described Godunov-type scheme is first-order accurate in space and time.

# 5 Gas dynamics toolbox

In this section the listings of two reusable MATLAB files required to simulate blast propagation in a fixed domain are given. Each function returns a structure with function handles.

The content of the `riemann_solver.m` M-file shown below defines a MATLAB function that solves the Riemann problem using the Newton–Raphson method. The acoustic approximation is taken as an initial guess in the iterative procedure. The function evaluates the combined Hugoniot–Poisson adiabatic curve, the gas state at the interface separating two adjacent cells, and the complete one-dimensional state of gas as a function of the coordinate $x$. The latter functionality is not used by the solver but provides a useful benchmark solution to compare with numerical results.

```
function rs = riemann_solver(gamma,tolerance,threshold)
% rs = riemann_solver(gamma,tolerance,threshold)
%
% Riemann solver for polytropic gas.
%
% Parameters:
%
%   gamma      - specific heat ratio
%   tolerance  - convergence tolerance for Newton's method [default: 1e-6]
%   threshold  - iteration threshold for Newton's method [default: 10]
%
% Function handles:
%
% r1 = rs.post_state_density(r0,p0,p1)
%
%   Calculates the post-state density r1 given the pre-state density r0,
%   pre-state pressure p0, and post-state pressure p1.
%
% v1 = rs.post_state_velocity(r0,p0,v0,p1)
%
%   Calculates the post-state velocity v1 given the pre-state density r0,
%   pre-state pressure p0, pre-state velocity v0, and post-state pressure
%   p1.
%
% [r,p,v] = rs.state(x,r1,p1,v1,r2,p2,v2)
%
%   Calculates density r, pressure p, velocity v at coordinate x pointing
%   from state 1 to state 2 separated by the interface x = 0.
%
% [r,a,p,vn,vt] = rs.interface_state(n,r1,a1,p1,v1,r2,a2,p2,v2)
%
%   Calculates density r, sound speed a, pressure p, normal velocity vn,
%   tangential velocity vt) at the interface where the unit normal vector n
%   points from state 1 to state 2.
%
% Examples:
%
%   *** Hugoniot-Poisson adiabatic curves ***
%
%   K = 3; N = 1000; gamma = linspace(1.5,2.5,K); p0 = 1; r0 = 1; v0 = 0;
%   p = linspace(0,5*p0,N); r = zeros(N,K); v = zeros(N,K);
%   info = cell(K,1);
%   for k = 1:K
%       rs = riemann_solver(gamma(k));
%       info{k} = sprintf('\\gamma = %g',gamma(k));
%       for i = 1:N
%           r(i,k) = rs.post_state_density(r0,p0,p(i));
%           v(i,k) = rs.post_state_velocity(r0,p0,v0,p(i));
%       end
%   end
%   subplot(2,1,1); plot(p,r); xlabel('Pressure'); ylabel('Density');
%   legend(info,'Location','Best');
%   subplot(2,1,2); plot(p,v); xlabel('Pressure'); ylabel('Velocity');
%   legend(info,'Location','Best');
%
%   *** Sod's shock tube ***
%
%   rs = riemann_solver(1.4);
%   [x,t] = ndgrid(linspace(-2,2,200),linspace(0,1,10));
%   [r,p,v] = rs.state(x./t,1,1,0,0.125,0.1,0);
%   figure; waterfall(x.',t.',r.'); view(30,40); title('Density');
%   figure; waterfall(x.',t.',p.'); view(30,40); title('Pressure');
%   figure; waterfall(x.',t.',v.'); view(30,40); title('Velocity');
%
% See also gas_dynamics_solver
%
% Copyright 2007-2008 Defence Science and Technology Organisation
```

```
%% Check input
if nargin < 3
    threshold = 10;
    if nargin < 2
        tolerance = 1.0e-6;
    end
end
if threshold < 1
    error('Iteration threshold must be positive: %d',threshold);
end
if tolerance < 1.0e-14
    error('Convergence tolerance must be positive: %g',tolerance);
end
if gamma <= 1.0
    error('Specific heat ratio must be greater than 1: %g',gamma);
end

%% Pre-calculate constants
gamma0 = 1.0/gamma;
gamma1 = 0.5*(gamma+1.0);
gamma2 = 0.5*(gamma-1.0);
gamma3 = gamma2/gamma1;
gamma4 = 1.0/gamma2;
gamma5 = gamma2*gamma0;

%% Define function handles
    function r1 = density(r0,p0,p1)
        if p1 >= p0
            r1 = r0*(p1+gamma3*p0)/(gamma3*p1+p0);
        elseif p1 > 0.0
            r1 = r0*(p1/p0)^gamma0;
        else
            r1 = 0.0;
        end
    end
    function v1 = velocity(r0,p0,v0,p1)
        if p1 >= p0
            v1 = v0+(p1-p0)/sqrt(r0*(gamma1*p1+gamma2*p0));
        elseif p1 > 0.0
            v1 = v0+gamma4*sqrt(gamma*p0/r0)*((p1/p0)^gamma5-1.0);
        else
            v1 = v0-gamma4*sqrt(gamma*p0/r0);
        end
    end
    function [v1,v1prime] = velocity_1jet(r0,a0,p0,v0,p1)
        if p1 >= p0
            t1 = gamma1*p1+gamma2*p0;
            t2 = sqrt(r0*t1);
            v1 = v0+(p1-p0)/t2;
            v1prime = (1.0-0.5*gamma1*(p1-p0)/t1)/t2;
        elseif p1 > 0.0
            t1 = (p1/p0)^gamma5;
            v1 = v0+a0*(t1-1.0)*gamma4;
            v1prime = gamma0*a0*t1/p1;
        else
            v1 = v0-gamma4*a0;
            v1prime = inf;
        end
    end
    function [r,a,p,v] = wave(x,r0,a0,p0,v0,r1,a1,p1,v1)
        if p1 > p0
            s = (r1*v1-r0*v0)/(r1-r0);
            if x > s
                r = r0;
                a = a0;
                p = p0;
            v = v0;
        elseif x < s
            r = r1;
            a = a1;
            p = p1;
            v = v1;
        else
            r = 0.5*(r0+r1);
            p = 0.5*(p0+p1);
            v = 0.5*(v0+v1);
            a = sqrt(gamma*p/r);
        end
    else
        if x >= v0+a0
            r = r0;
            a = a0;
            p = p0;
            v = v0;
        elseif x <= v1+a1
            r = r1;
            a = a1;
            p = p1;
            v = v1;
        else
            a = a0-gamma3*(v0+a0-x);
            r = r0*(a/a0)^gamma4;
            p = p0*(r/r0)^gamma;
            v = x-a;
        end
    end
end
function [p,v] = newton_raphson(r1,a1,p1,v1,r2,a2,p2,v2)
    w1 = 1.0/(r1*a1);
    w2 = 1.0/(r2*a2);
    p = max((w1*p1+w2*p2+v1-v2)/(w1+w2),tolerance);
    i = 0;
    e = 1.0+tolerance;
    while e > tolerance
        i = i+1;
        if i > threshold
            error('Reached iteration threshold: %g',e);
        end
        [u,uprime] = velocity_1jet(r1,a1,p1,-v1,p);
        [v,vprime] = velocity_1jet(r2,a2,p2,v2,p);
        e = (v+u)/(vprime+uprime);
        p = max(p-e,tolerance);
        e = abs(e/p);
    end
    v = 0.5*(v-u);
end
function [r,p,v] = state(x,r1,p1,v1,r2,p2,v2)
    a1 = sqrt(gamma*p1/r1);
    a2 = sqrt(gamma*p2/r2);
    [p0,v0] = newton_raphson(r1,a1,p1,v1,r2,a2,p2,v2);
    r = zeros(size(x));
    p = zeros(size(x));
    v = zeros(size(x));
    for i = 1:numel(x)
        if x(i) < v0
            r0 = density(r1,p1,p0);
            a0 = sqrt(gamma*p0/r0);
            [r(i),a,p(i),v(i)] = wave(-x(i),r1,a1,p1,-v1,r0,a0,p0,-v0);
            v(i) = -v(i);
        elseif x(i) > v0
            r0 = density(r2,p2,p0);
            a0 = sqrt(gamma*p0/r0);
            [r(i),a,p(i),v(i)] = wave(x(i),r2,a2,p2,v2,r0,a0,p0,v0);
            else
                r0 = density(r1,p1,p0);
                a0 = sqrt(gamma*p0/r0);
                [rr1,a,pp1,vv1] = wave(-x(i),r1,a1,p1,-v1,r0,a0,p0,-v0);
                vv1 = -vv1;
                r0 = density(r2,p2,p0);
                a0 = sqrt(gamma*p0/r0);
                [rr2,a,pp2,vv2] = wave(x(i),r2,a2,p2,v2,r0,a0,p0,v0);
                r(i) = 0.5*(rr1+rr2);
                p(i) = 0.5*(pp1+pp2);
                v(i) = 0.5*(vv1+vv2);
            end
        end
    end
    function [r,a,p,vn,vt] = interface_state(n,r1,a1,p1,v1,r2,a2,p2,v2)
        v1n = dot(v1,n);
        v2n = dot(v2,n);
        v1t = v1-v1n*n;
        v2t = v2-v2n*n;
        [p0,v0] = newton_raphson(r1,a1,p1,v1n,r2,a2,p2,v2n);
        switch sign(v0)
            case 1
                r0 = density(r1,p1,p0);
                a0 = sqrt(gamma*p0/r0);
                [r,a,p,vn] = wave(0.0,r1,a1,p1,-v1n,r0,a0,p0,-v0);
                vn = -vn;
                vt = v1t;
            case -1
                r0 = density(r2,p2,p0);
                a0 = sqrt(gamma*p0/r0);
                [r,a,p,vn] = wave(0.0,r2,a2,p2,v2n,r0,a0,p0,v0);
                vt = v2t;
            otherwise
                r0 = density(r1,p1,p0);
                a0 = sqrt(gamma*p0/r0);
                [rr1,aa1,pp1,vn1] = wave(0.0,r1,a1,p1,-v1n,r0,a0,p0,-v0);
                vn1 = -vn1;
                r0 = density(r2,p2,p0);
                a0 = sqrt(gamma*p0/r0);
                [rr2,aa2,pp2,vn2] = wave(0.0,r2,a2,p2,v2n,r0,a0,p0,v0);
                r = 0.5*(rr1+rr2);
                p = 0.5*(pp1+pp2);
                vn = 0.5*(vn1+vn2);
                vt = 0.5*(v1t+v2t);
                a = sqrt(gamma*p/r);
        end
    end

%% Publish the interface
rs.state = @state;
rs.interface_state = @interface_state;
rs.post_state_density = @density;
rs.post_state_velocity = @velocity;
end
```

The content of the `gas_dynamics_solver.m` M-file shown below defines a MATLAB function that implements a gas dynamics solver. The user needs to populate a data structure and pass it to the solver. The solver is generic as it does not use any specific mesh. Instead, only a facet-to-cell connectivity matrix, an array of facet vector area elements and an array of control volumes are used to define geometry. Moreover, this function does not depend on the `riemann_solver.m` function as it works with a function handle for interface state calculation. The `riemann_solver.m` provides such a handle, but other implementations can be equally used, such as the approximate Roe solver [Roe 1981].

```
function gds = gas_dynamics_solver
% gds = gas_dynamics_solver
%
% Gas dynamics solver.
%
% Function handles:
%
% model = gds.initialise(model)
%
%   Validates the model structure, allocates auxiliary fields and populates
%   the conserved variables.
%
% model = gds.update_rates(model)
%
%   Calculates the interface states from the current physical variables and
%   updates the rates of the conserved variables.
%
% model = gds.update_states(model)
%
%   Updates the physical variables from the conserved variables.
%   Should be always called after advancing the time step and appying
%   boundary conditions that must affect the conserved variables only.
%
% The model structure must contain the following fields:
%
%   model.gamma   - polytropic index
%   model.state   - interface state sampling handle
%   model.fc      - facet-to-cell connectivity matrix
%   model.ds      - vector area elements
%   model.dv      - volume elements
%   model.r       - density
%   model.p       - pressure
%   model.v       - velocity
%
% The following auxiliary fields will be allocated:
%
%   model.gamma1  - model.gamma - 1
%   model.da      - facet area elements
%   model.n       - facet unit normal vectors
%   model.a       - sound speed
%   model.e       - total energy
%   model.u       - momentum
%   model.r_rate  - density rate
%   model.e_rate  - total energy rate
%   model.u_rate  - momentum rate
%   model.ir      - interface density
%   model.ip      - interface pressure
%   model.ia      - interface sound speed
%   model.ivn     - interface normal component of velocity
%
% Examples:
%
%   *** Sod's shock tube ***
%
%   model.gamma = 1.4; rs = riemann_solver(model.gamma);
%   model.state = rs.interface_state; gds = gas_dynamics_solver;
%   N = 200; L = 1.0; dx = L/N; x = -dx/2:dx:1+dx/2;
%   model.dv = dx*ones(N+2,1); model.ds = ones(N+1,1);
%   model.fc = [1:N+1;2:N+2].'; model.r = zeros(N+2,1);
%   model.p = zeros(N+2,1); model.v = zeros(N+2,1);
%   boundary_cells = [2,N+1]; ghost_cells = [1,N+2];
%   for i = 1:N+2
%       if x(i) < L/2
%           model.r(i) = 1.0; model.p(i) = 1.0;
%       else
%           model.r(i) = 0.125; model.p(i) = 0.1;
%       end
%   end
%   model = gds.initialise(model);
%   t = 0; tf = 0.25;
%   while t < tf
%       model = gds.update_rates(model);
%       dt = dx/max(model.ia+abs(model.ivn));
%       dt = min(dt,tf-t); t = t+dt;
%       model.r = model.r+model.r_rate*dt;
%       model.e = model.e+model.e_rate*dt;
%       model.u = model.u+model.u_rate*dt;
%       model.r(ghost_cells) = model.r(boundary_cells);
%       model.e(ghost_cells) = model.e(boundary_cells);
%       model.u(ghost_cells) = model.u(boundary_cells);
%       model = gds.update_states(model);
%       subplot(3,1,1);
%       plot(x(2:end-1),model.r(2:end-1),'.');
%       title(sprintf('Density at t = %g',t));
%       subplot(3,1,2);
%       plot(x(2:end-1),model.p(2:end-1),'.');
%       title(sprintf('Pressure at t = %g',t));
%       subplot(3,1,3);
%       plot(x(2:end-1),model.v(2:end-1),'.');
%       title(sprintf('Velocity at t = %g',t));
%       drawnow;
%   end
%
% See also riemann_solver
%
% Copyright 2007-2008 Defence Science and Technology Organisation

%% Define function handles
    function model = initialise(model)
        % Check required fields
        if ~isfield(model,'gamma')
            error('Undefined polytropic index: model.gamma');
        end
        if ~isfield(model,'state')
            error('Undefined interface state sampling handle: model.state');
        end
        if ~isa(model.state,'function_handle')
            error('Field model.state must be a function handle');
        end
        if ~isfield(model,'fc')
            error('Undefined facet-to-cell connectivity matrix: model.fc');
        end
        if ~isfield(model,'ds')
            error('Undefined vector area elements: model.ds');
        end
        if ~isfield(model,'dv')
            error('Undefined volume elements: model.dv');
        end
        if ~isfield(model,'r')
            error('Undefined density: model.r');
        end
        if ~isfield(model,'p')
            error('Undefined pressure: model.p');
        end
        if ~isfield(model,'v')
            error('Undefined velocity: model.v');
        end
        % Check consistency
        if size(model.ds,2) ~= size(model.v,2)
            error('Unequal number of columns of model.ds and model.v');
        end
        model.DIM = size(model.v,2);
        if model.DIM < 1 || model.DIM > 3
            error('Invalid space dimension: %d',model.DIM);
    end
    if size(model.fc,2) ~= 2
        error('Connectivity matrix must have two columns');
    end
    if size(model.dv,2) ~= 1
        error('Control volumes must form a column vector');
    end
    if size(model.r,2) ~= 1
        error('Density field must be a column vector');
    end
    if size(model.p,2) ~= 1
        error('Pressure field must be a column vector');
    end
    if size(model.dv,1) ~= size(model.r,1)
        error('Unequal number of rows of model.dv and model.r');
    end
    if size(model.dv,1) ~= size(model.p,1)
        error('Unequal number of rows of model.dv and model.p');
    end
    if size(model.dv,1) ~= size(model.v,1)
        error('Unequal number of rows of model.dv and model.v');
    end
    % Initialise
    model.gamma1 = model.gamma-1.0;
    model.da = sqrt(dot(model.ds,model.ds,2));
    model.n(:,1) = model.ds(:,1)./model.da;
    if model.DIM > 1
        model.n(:,2) = model.ds(:,2)./model.da;
        if model.DIM > 2
            model.n(:,3) = model.ds(:,3)./model.da;
        end
    end
    model.a = zeros(size(model.r));
    model.e = zeros(size(model.p));
    model.u = zeros(size(model.v));
    model.r_rate = zeros(size(model.r));
    model.e_rate = zeros(size(model.e));
    model.u_rate = zeros(size(model.u));
    model.ir = zeros(size(model.ds,1),1);
    model.ip = zeros(size(model.ds,1),1);
    model.ia = zeros(size(model.ds,1),1);
    model.ivn = zeros(size(model.ds,1),1);
    model.a = sqrt(model.gamma*model.p./model.r);
    model.u(:,1) = model.r.*model.v(:,1);
    if model.DIM > 1
        model.u(:,2) = model.r.*model.v(:,2);
        if model.DIM > 2
            model.u(:,3) = model.r.*model.v(:,3);
        end
    end
    model.e = model.p/model.gamma1+0.5*dot(model.u,model.v,2);
end
function model = update_rates(model)
    model.r_rate = zeros(size(model.r_rate));
    model.e_rate = zeros(size(model.e_rate));
    model.u_rate = zeros(size(model.u_rate));
    for k = 1:size(model.fc,1)
        i1 = model.fc(k,1);
        i2 = model.fc(k,2);
        [r,a,p,vn,vt] = model.state(model.n(k,:),...
            model.r(i1),model.a(i1),model.p(i1),model.v(i1,:),...
            model.r(i2),model.a(i2),model.p(i2),model.v(i2,:));
        model.ir(k) = r;
        model.ia(k) = a;
        model.ip(k) = p;
        model.ivn(k) = vn;
        v = vt+vn*model.n(k,:);
```

```
            r_flux = r*vn*model.da(k);
            e_flux = r_flux*(a*a/model.gamma1+0.5*dot(v,v,2));
            u_flux = r_flux*v+p*model.ds(k,:);
            model.r_rate(i1) = model.r_rate(i1)-r_flux;
            model.r_rate(i2) = model.r_rate(i2)+r_flux;
            model.e_rate(i1) = model.e_rate(i1)-e_flux;
            model.e_rate(i2) = model.e_rate(i2)+e_flux;
            model.u_rate(i1,:) = model.u_rate(i1,:)-u_flux;
            model.u_rate(i2,:) = model.u_rate(i2,:)+u_flux;
        end
        model.r_rate = model.r_rate./model.dv;
        model.e_rate = model.e_rate./model.dv;
        model.u_rate(:,1) = model.u_rate(:,1)./model.dv;
        if model.DIM > 1
            model.u_rate(:,2) = model.u_rate(:,2)./model.dv;
            if model.DIM > 2
                model.u_rate(:,3) = model.u_rate(:,3)./model.dv;
            end
        end
    end
    function model = update_states(model)
        model.v(:,1) = model.u(:,1)./model.r;
        if model.DIM > 1
            model.v(:,2) = model.u(:,2)./model.r;
            if model.DIM > 2
                model.v(:,3) = model.u(:,3)./model.r;
            end
        end
        model.p = model.gamma1*(model.e-0.5*dot(model.u,model.v,2));
        model.a = sqrt(model.gamma*model.p./model.r);
    end

%% Publish the interface
gds.initialise = @initialise;
gds.update_rates = @update_rates;
gds.update_states = @update_states;
end
```

# 6 Usage examples

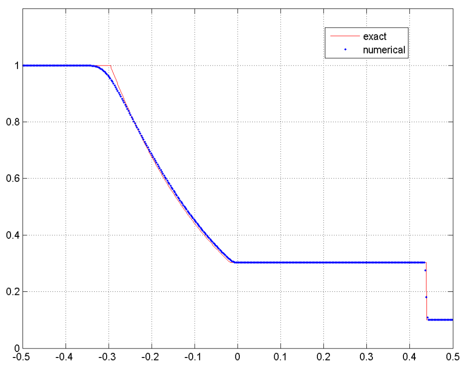
## 6.1 Sod's shock tube

The exact similarity solution to Sod's shock tube [Sod 1978] specified by the following initial piecewise-constant state of gas

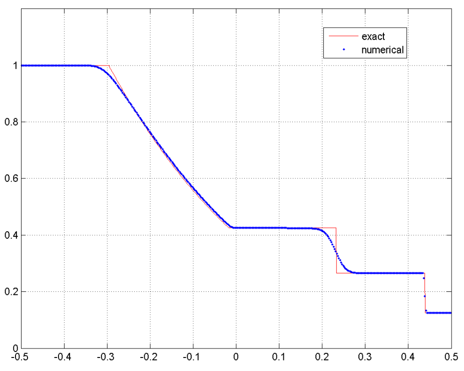
$$(\rho,\, p,\, v) = \begin{cases} (1.0,\, 1.0,\, 0.0) & (x < 0) \\ (0.125,\, 0.1,\, 0.0) & (x > 0) \end{cases} \tag{40}$$

provides a useful benchmark for gas dynamics solvers.

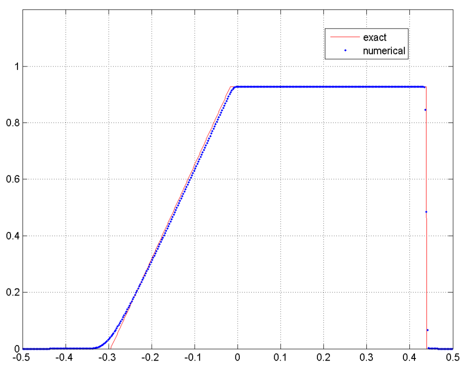
The following MATLAB script simulates the benchmark problem of ideal polytropic gas with specific heat ratio $\gamma = 1.4$ on a uniform mesh consisting of 400 control cells. The numerical results are displayed in Figure 2. It is seen that the numerical solution and the exact benchmark solution are in good agreement. Note that, due to numerical viscosity, the contact discontinuity is smeared out to some extent whereas the shock front position is captured quite accurately.



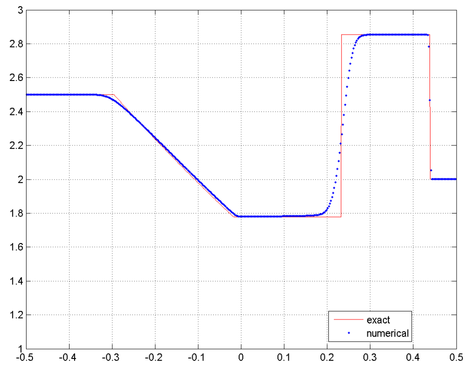
(i) Pressure.



(ii) Density.



(iii) Velocity.



(iv) Internal energy.

***Figure 2:*** *Gas state in Sod's shock tube at time* $t = 0.25$.

```
%% Sod's shock tube simulation
x1 = -0.5; r1 = 1.0; p1 = 1.0; v1 = 0.0;
x2 = 0.5; r2 = 0.125; p2 = 0.1; v2 = 0.0;
tf = 0.25; NX = 400;

%% Benchmark solution
model.gamma = 1.4; rs = riemann_solver(model.gamma);
model.state = rs.interface_state;
X = linspace(x1,x2,1000); [R,P,V] = rs.state(X/tf,r1,p1,v1,r2,p2,v2);

%% Generate a uniform mesh
dx = (x2-x1)/NX; x = (x1-dx/2:dx:x2+dx/2).';
model.fc = [1:NX+1;2:NX+2].'; model.ds = ones(NX+1,1);
model.dv = dx*ones(NX+2,1); B = [2,NX+1]; G = [1,NX+2]; I = 2:NX;

%% Run simulation
disp('*** Sod''s shock tube ***');
t = 0.0;
[model.r,model.p,model.v] = rs.state(x/t,r1,p1,v1,r2,p2,v2);
model.v = zeros(NX+2,1); gds = gas_dynamics_solver;
model = gds.initialise(model);
while t < tf
    model = gds.update_rates(model);
    dt = dx/max(model.ia+abs(model.ivn));
    dt = min(dt,tf-t); t = t+dt; disp(sprintf('t = %g',t));
    model.r = model.r+model.r_rate*dt;
    model.e = model.e+model.e_rate*dt;
    model.u = model.u+model.u_rate*dt;
    model.r(G) = model.r(B);
    model.e(G) = model.e(B);
    model.u(G) = -model.u(B);
    model = gds.update_states(model);
end

%% Plot the final state
plot(X,R,'r-',x,model.r,'b.'); grid on;
axis([x1,x2,0,1.2]); legend('exact','numerical','Location','Best');
print -dpng sod_shock_tube_density;
plot(X,P./R/model.gamma1,'r-',x,model.p./model.r/model.gamma1,'b.'); grid on;
axis([x1,x2,1.0,3.0]); legend('exact','numerical','Location','Best');
print -dpng sod_shock_tube_energy;
plot(X,P,'r-',x,model.p,'b.'); grid on;
axis([x1,x2,0,1.2]); legend('exact','numerical','Location','Best');
print -dpng sod_shock_tube_pressure;
plot(X,V,'r-',x,model.v,'b.'); grid on;
axis([x1,x2,0,1.2]); legend('exact','numerical','Location','Best');
print -dpng sod_shock_tube_velocity;
```

## 6.2 Rotationally symmetric blast

Another interesting one-dimensional example that can be used as a benchmark for testing multi-dimensional code is a rotationally symmetric blast in two or three dimensions. All the gas state variables are functions of the radial coordinate $r$ and time $t$ and velocity has only the radial non-zero component $v$. The governing equations are as follows

$$\frac{\partial\rho}{\partial t}+\frac{\partial(\rho\,v)}{\partial r}+\frac{\nu\,\rho\,v}{r}=0\,, \tag{41}$$

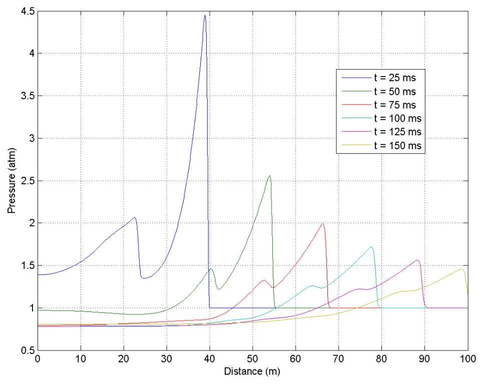
$$\frac{\partial(\rho\,v)}{\partial t}+\frac{\partial\big[\big(\rho\,v^2+p\big)\;v\big]}{\partial r}+\frac{\nu\,\rho\,v^2}{r}=0\,, \tag{42}$$

$$\frac{\partial\varepsilon}{\partial t}+\frac{\partial[(\varepsilon+p)\;v]}{\partial r}+\frac{\nu\;(\varepsilon+p)\;v}{r}=0\,, \tag{43}$$

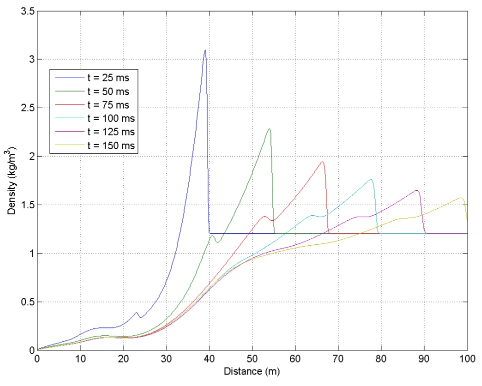
where $\nu=2$ for rotationally symmetric three-dimensional blast (described in the spherical coordinates) and $\nu=1$ for rotationally symmetric two-dimensional blast (described in the polar coordinates). Note that the case $\nu=0$ corresponds to the one-dimensional problem describing shock propagation in a tube, such as the one considered in the previous section. For nonzero $\nu$, equations (41)–(43) contain geometric source terms which can be easily incorporated into the numerical scheme after the calculation of the gas state rates due to fluxes alone.

The following script simulates the propagation of blast waves from a spherical charge ($\nu=2$). The results of the numerical simulation are presented in Figure 3. It is seen the formation of a secondary shock wave following the main shock [Brode 1959].

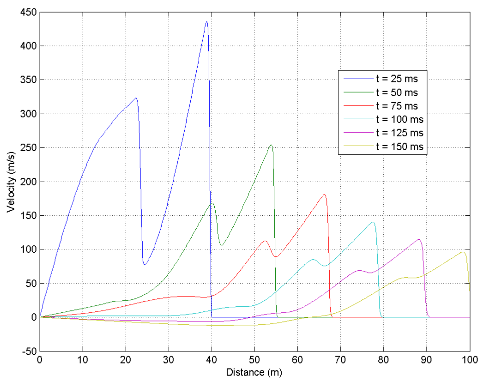
```
%% Spherically symmetric blast
clear; project_name = 'radial_explosion'; choice = 1;
if exist(sprintf('%s.mat',project_name),'file')
    choice = menu('Re-run?','Yes','No, just post-process','Cancel');
    pause(0.1);
```



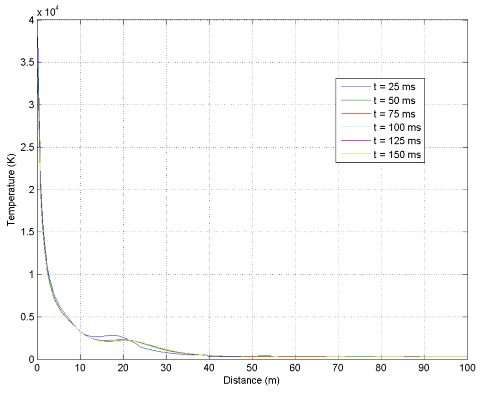
(i) Pressure.



(ii) Density.



(iii) Velocity.



(iv) Temperature.

***Figure 3:*** *Blast waves from a spherical charge.*

```
end
switch choice
    case 1
        % Specify parameters
        model.gamma = 1.4; rs = riemann_solver(model.gamma,1.0e-9,100);
        model.state = rs.interface_state;
        nu = 2; L = 100.0; w = 1.0; R = 287.0;
        p0 = 1.01e5; r0 = 1.204; p1 = 1.0e5*p0; r1 = 1.0e3;
        NX = 1000; CFL = 0.9;
        % Generate a uniform mesh
        dx = L/NX; x = (-dx/2:dx:L+dx/2).'; z = -nu./x;
        model.dv = dx*ones(NX+2,1); model.ds = ones(NX+1,1);
        model.fc = [1:NX+1;2:NX+2].';
        B1 = 2; G1 = 1; B2 = NX+1; G2 = NX+2;
        % Initial conditions
        model.r = zeros(NX+2,1);
        model.p = zeros(NX+2,1);
        model.v = zeros(NX+2,1);
        for i = 1:NX+2
            if x(i) < w
                model.r(i) = r1; model.p(i) = p1;
            else
                model.r(i) = r0; model.p(i) = p0;
            end
            end
            % Allocate variables for post-processing
            times = [0.025,0.05,0.075,0.1,0.125,0.15]; K = length(times);
            pressures = zeros(NX,K);
            velocities = zeros(NX,K);
            densities = zeros(NX,K);
            temperatures = zeros(NX,K);
            I = 2:NX+1; x = x(I);
            % Run simulation
            disp('*** Spherically symmetric blast ***');
            gs = gas_dynamics_solver; model = gs.initialise(model);
            t = 0.0;
            for k = 1:K
                tf = times(k);
                while t < tf
                    model = gs.update_rates(model);
                    dt = CFL*dx/max(abs(model.ivn)+model.ia);
                    dt = min(dt,tf-t); t = t+dt;
                    model.r = model.r+model.r_rate*dt;
                    model.e = model.e+model.e_rate*dt;
                    model.u = model.u+model.u_rate*dt;
                    model = gs.update_states(model);
                    model.r_rate = z.*model.u;
                    model.e_rate = model.r_rate.*(model.e+model.p)./model.r;
                    model.u_rate = model.r_rate.*model.v;
                    model.r = model.r+model.r_rate*dt;
                    model.e = model.e+model.e_rate*dt;
                    model.u = model.u+model.u_rate*dt;
                    model.r(G1) = model.r(B1); % reflective BC
                    model.e(G1) = model.e(B1);
                    model.u(G1) = -model.u(B1);
                    model.r(G2) = model.r(B2); % open-end BC
                    model.e(G2) = model.e(B2);
                    model.u(G2) = model.u(B2);
                    model = gs.update_states(model);
                    plot(x,model.p(I),'k.');
                    title(sprintf('Pressure at t = %g',t));
                    drawnow;
                end
                pressures(:,k) = model.p(I);
                velocities(:,k) = model.v(I);
                densities(:,k) = model.r(I);
                temperatures(:,k) = model.p(I)./model.r(I)/R;
            end
            save(project_name);
        case 2
            load(project_name);
        otherwise
            return;
    end
    %% Generate images
    info = cell(size(times));
    for k = 1:K
        info{k} = sprintf('t = %g ms',1000.0*times(k));
    end
    figure; plot(x,pressures/p0); grid on;
    xlabel('Distance (m)'); ylabel('Pressure (atm)');
    legend(info,'Location','Best');
    eval(sprintf('print -dpng %s_pressure',project_name));
    figure; plot(x,velocities); grid on;
    xlabel('Distance (m)'); ylabel('Velocity (m/s)');
    legend(info,'Location','Best');
    eval(sprintf('print -dpng %s_velocity',project_name));
    figure; plot(x,densities); grid on;
    xlabel('Distance (m)'); ylabel('Density (kg/m^3)');
    legend(info,'Location','Best');
    eval(sprintf('print -dpng %s_density',project_name));
```

```
figure; plot(x,temperatures); grid on;
xlabel('Distance (m)'); ylabel('Temperature (K)');
legend(info,'Location','Best');
eval(sprintf('print -dpng %s_temperature',project_name));
```

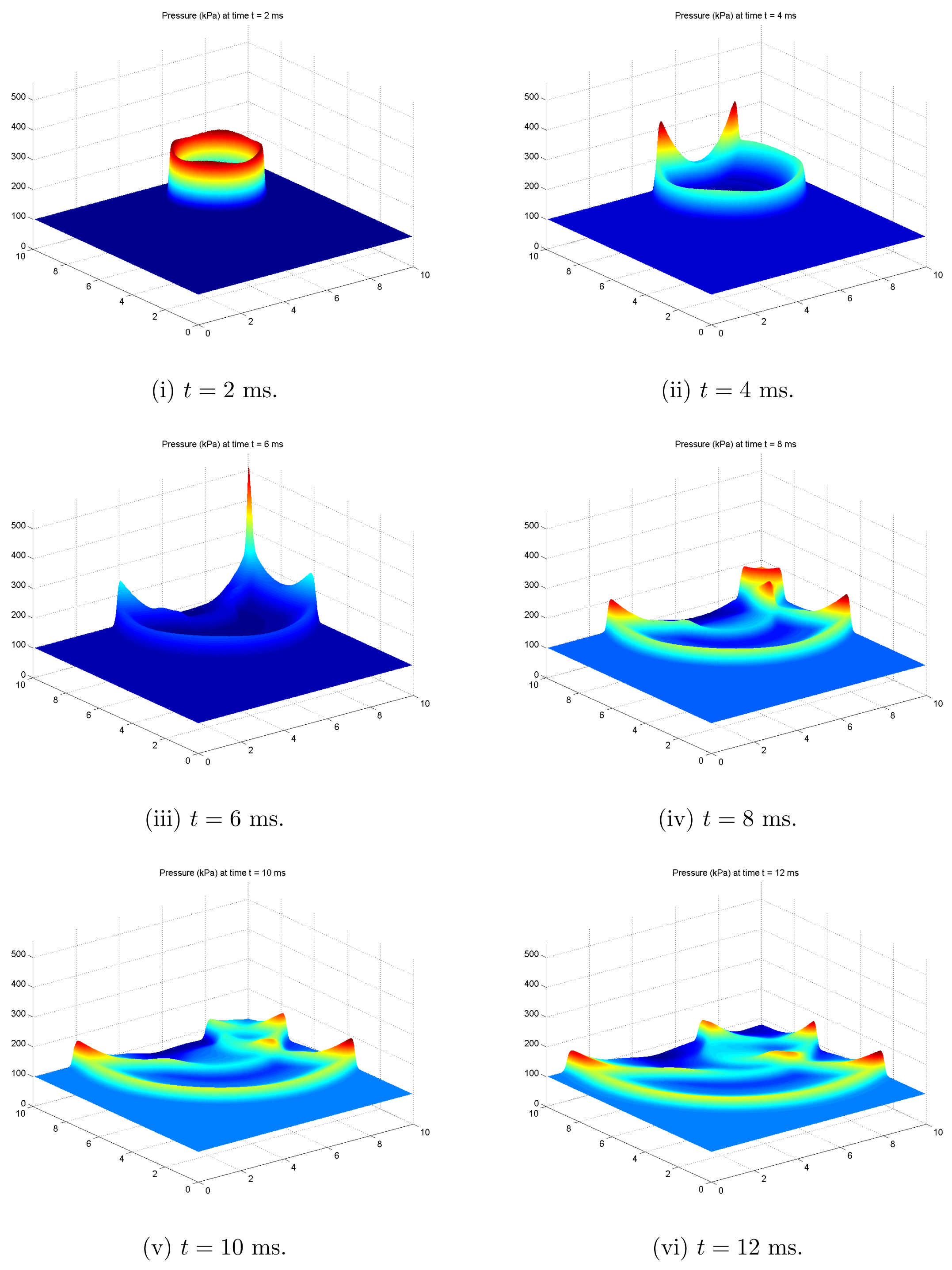
## 6.3 Two-dimensional blast propagation

Consider a two-dimensional explosion in air occupying a square domain with a side length of 10 m at initial atmospheric pressure of 101 kPa (1 atmosphere) and temperature of 20°C. The blast is modelled by a localised region of 10-atmosphere over-pressure and temperature of 2000°C.

The following MATLAB script simulates this problem on a uniform mesh. The pressure and temperature fields for increasing times are presented in Figures 4 and 5 respectively.

```
%% 2D blast propagation in a square domain
project_name = 'blast2d';
disp(sprintf('Project name: %s',project_name));
gds = gas_dynamics_solver;
R = 287.0; Kelvin0 = 273.0;
T0 = Kelvin0+20.0; p0 = 1.01e5; r0 = p0/T0/R;
T1 = Kelvin0+2000.0; p1 = 11*p0; r1 = p1/T1/R;
CFL = 0.5; save_time = 0.002; kmax = 6;

%% Pre-process if necessary
k = 0;
if ~exist(sprintf('%s%d.mat',project_name,k),'file')
    % Generate mesh
    s.LX = 10.0; s.LY = 10.0; s.NX = 250; s.NY = 250;
    s.map = @(ix,iy) sub2ind([s.NX+2,s.NY+2],ix,iy);
    dx = s.LX/s.NX; dy = s.LY/s.NY; s.dx = min([dx,dy]);
    s.x = -dx/2:dx:s.LX+dx/2; s.y = -dy/2:dy:s.LY+dy/2;
    NC = (s.NX+2)*(s.NY+2); NF = (s.NX+1)*s.NY+s.NX*(s.NY+1);
    disp(sprintf('%d facets, %d cells',NF,NC-4));
    s.dv = (dx*dy)*ones(NC,1); s.fc = zeros(NF,2); s.ds = zeros(NF,2);
    s.BE = zeros(s.NY,1); s.GE = zeros(s.NY,1); % east BC
    s.BW = zeros(s.NY,1); s.GW = zeros(s.NY,1); % west BC
    s.BS = zeros(s.NX,1); s.GS = zeros(s.NX,1); % south BC
    s.BN = zeros(s.NX,1); s.GN = zeros(s.NX,1); % north BC
    i = 0; % facet index
    for ix = 1:s.NX+1
        for iy = 2:s.NY+1
            i = i+1; s.ds(i,:) = [dy,0];
            s.fc(i,1) = s.map(ix,iy);
            s.fc(i,2) = s.map(ix+1,iy);
        end
    end
    for iy = 1:s.NY+1
        for ix = 2:s.NX+1
            i = i+1; s.ds(i,:) = [0,dx];
            s.fc(i,1) = s.map(ix,iy);
            s.fc(i,2) = s.map(ix,iy+1);
        end
    end
    for iy = 2:s.NY+1
        s.BE(iy-1) = s.map(2,iy); s.GE(iy-1) = s.map(1,iy);
        s.BW(iy-1) = s.map(s.NX+1,iy); s.GW(iy-1) = s.map(s.NX+2,iy);
    end
    for ix = 2:s.NX+1
        s.BS(ix-1) = s.map(ix,2); s.GS(ix-1) = s.map(ix,1);
        s.BN(ix-1) = s.map(ix,s.NY+1); s.GN(ix-1) = s.map(ix,s.NY+2);
    end
```

```
        % Initial conditions
        xa = 7.0; ya = 8.0; a = 0.5; b = 100.0;
        density = @(x,y)...
            0.5*((r1+r0)+(r1-r0)*tanh(b*(a-sqrt((x-xa)^2+(y-ya)^2))));
        pressure = @(x,y)...
            0.5*((p1+p0)+(p1-p0)*tanh(b*(a-sqrt((x-xa)^2+(y-ya)^2))));
        s.r = zeros(NC,1); s.p = zeros(NC,1); s.v = zeros(NC,2);
        for ix = 1:s.NX+2
            for iy = 1:s.NY+2
                i = s.map(ix,iy);
                s.r(i) = density(s.x(ix),s.y(iy));
                s.p(i) = pressure(s.x(ix),s.y(iy));
            end
        end
        % Specify Riemann solver
        s.gamma = 1.4; rs = riemann_solver(s.gamma);
        s.state = rs.interface_state; s.t = 0.0; s = gds.initialise(s);
        eval(sprintf('save %s%d -struct s',project_name,k));
    end

    %% Find the latest restart file
    while exist(sprintf('%s%d.mat',project_name,k),'file')
        k = k+1;
    end
    k = k-1;

    %% Load start/restart file
    s = load(sprintf('%s%d',project_name,k));
    while k < kmax
        elapsed_time = cputime;
        k = k+1;
        tf = s.t + save_time;
        while s.t < tf
            s = gds.update_rates(s);
            dt = CFL*s.dx/max(s.ia+abs(s.ivn));
            dt = min(dt,tf-s.t); s.t = s.t+dt;
            disp(sprintf('t = %g',s.t));
            s.r = s.r+s.r_rate*dt;
            s.e = s.e+s.e_rate*dt;
            s.u = s.u+s.u_rate*dt;
            % reflective east BC
            s.r(s.GE) = s.r(s.BE); s.e(s.GE) = s.e(s.BE);
            s.u(s.GE,1) = -s.u(s.BE,1); s.u(s.GE,2) = s.u(s.BE,2);
            % reflective west BC
            s.r(s.GW) = s.r(s.BW); s.e(s.GW) = s.e(s.BW);
            s.u(s.GW,1) = -s.u(s.BW,1); s.u(s.GW,2) = s.u(s.BW,2);
            % reflective south BC
            s.r(s.GS) = s.r(s.BS); s.e(s.GS) = s.e(s.BS);
            s.u(s.GS,1) = s.u(s.BS,1); s.u(s.GS,2) = -s.u(s.BS,2);
            % reflective north BC
            s.r(s.GN) = s.r(s.BN); s.e(s.GN) = s.e(s.BN);
            s.u(s.GN,1) = s.u(s.BN,1); s.u(s.GN,2) = -s.u(s.BN,2);
            % synchronise physical variables
            s = gds.update_states(s);
        end
        elapsed_time = cputime-elapsed_time;
        disp(sprintf('CPU time: %g secs / %g mins / %g hours',...
            elapsed_time,elapsed_time/60,elapsed_time/3600));
        eval(sprintf('save %s%d -struct s',project_name,k));
    end

    %% Post process
    disp('Generating PNG images...');
    set(clf,'Renderer','ZBuffer');
    k = 0;
    while exist(sprintf('%s%d.mat',project_name,k),'file')
        s = load(sprintf('%s%d',project_name,k));
    [x,y] = ndgrid(s.x,s.y); z = zeros(size(x));
    for ix = 2:s.NX+1
        for iy = 2:s.NY+1
            i = s.map(ix,iy);
            z(ix,iy) = s.p(i)/1000;
        end
    end
    surf(x(2:end-1,2:end-2),y(2:end-1,2:end-2),z(2:end-1,2:end-2));
    title(sprintf('Pressure (kPa) at time t = %g ms',1000*s.t));
    axis([0,s.LX,0,s.LY,0,p1/2000]); shading interp;
    eval(sprintf('print -dpng %s%d_pressure',project_name,k));
    for ix = 2:s.NX+1
        for iy = 2:s.NY+1
            i = s.map(ix,iy);
            z(ix,iy) = s.p(i)/s.r(i)/R;
        end
    end
    surf(x(2:end-1,2:end-2),y(2:end-1,2:end-2),z(2:end-1,2:end-2));
    title(sprintf('Temperature (K) at time t = %g ms',1000*s.t));
    axis([0,s.LX,0,s.LY,0,T1]); shading interp;
    eval(sprintf('print -dpng %s%d_temperature',project_name,k));
    k = k+1;
end
```



(i) $t = 2$ ms.

(ii) $t = 4$ ms.

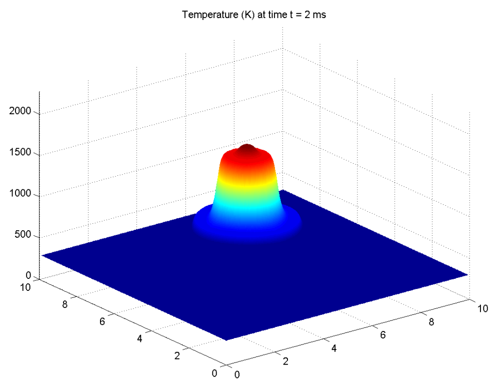
(iii) $t = 6$ ms.

(iv) $t = 8$ ms.

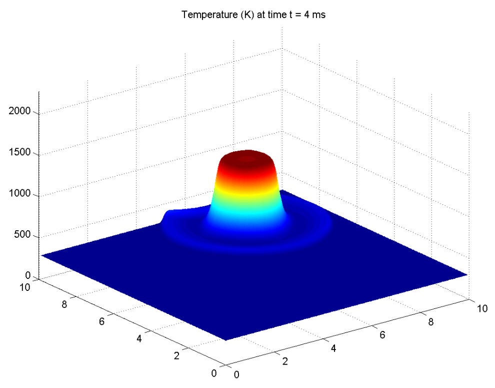
(v) $t = 10$ ms.

(vi) $t = 12$ ms.

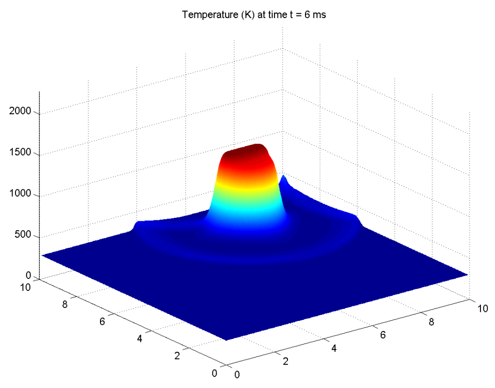
***Figure 4:*** *Two-dimensional blast propagation; pressure field surfaces.*



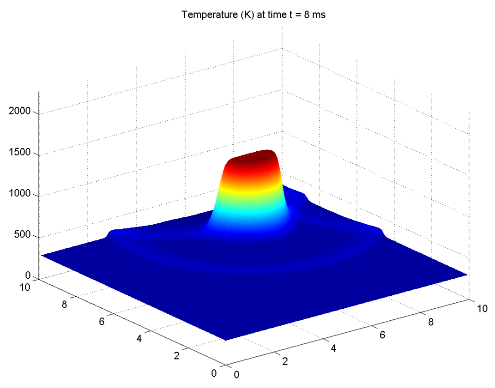
(i) $t = 2$ ms.



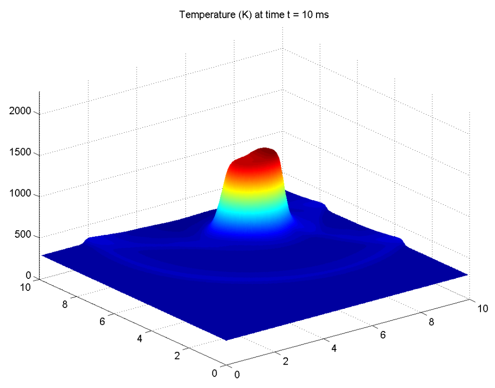
(ii) $t = 4$ ms.



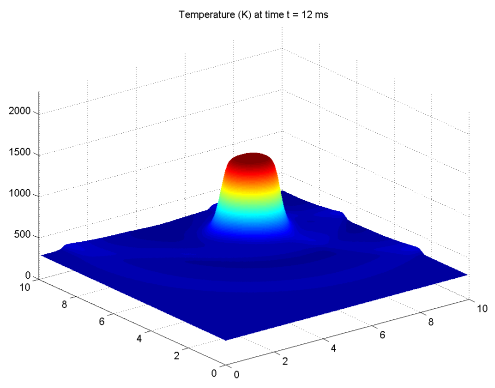
(iii) $t = 6$ ms.



(iv) $t = 8$ ms.



(v) $t = 10$ ms.



(vi) $t = 12$ ms.

***Figure 5:*** *Two-dimensional blast propagation; temperature field surfaces.*

## 6.4 Three-dimensional blast propagation

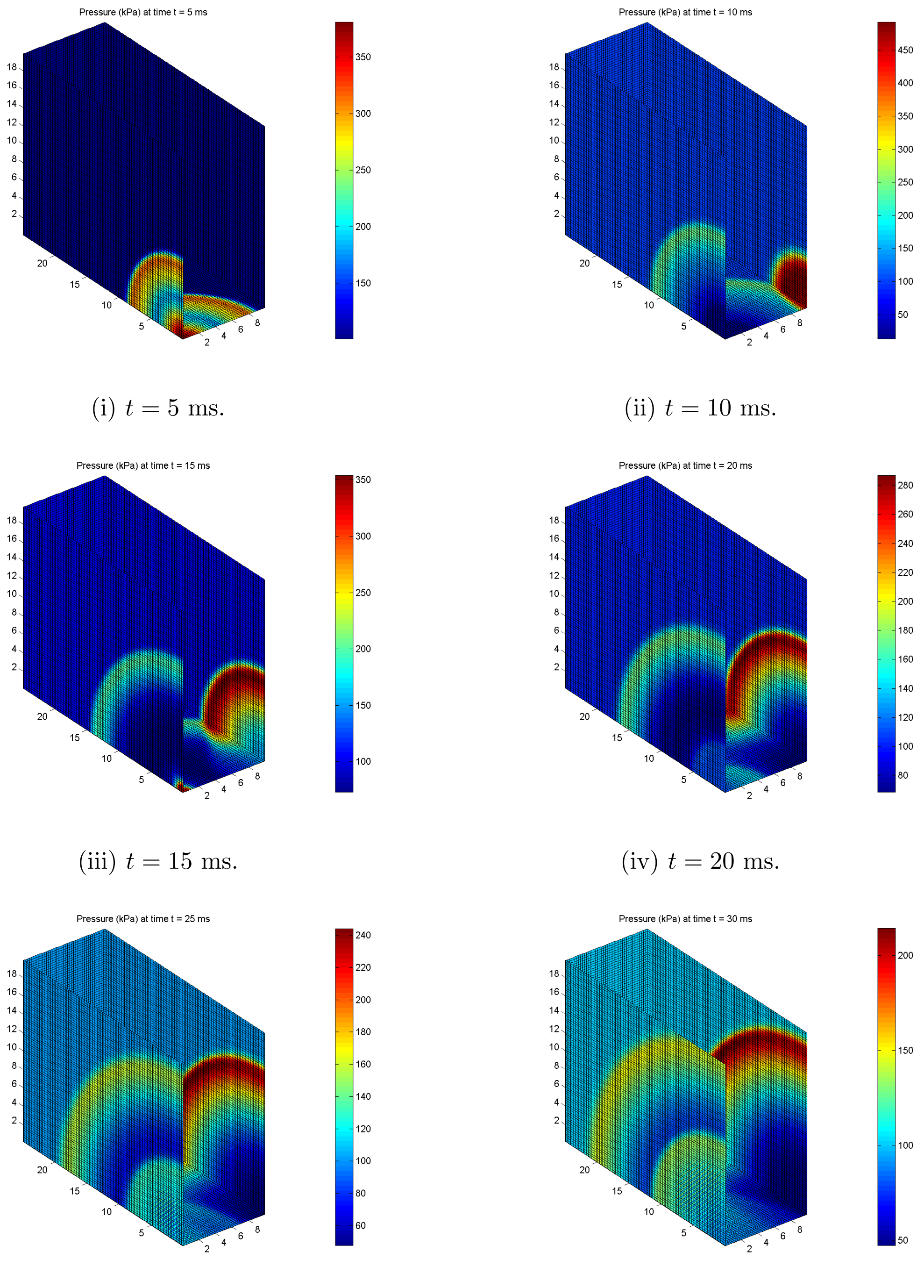
Consider three-dimensional explosion in air occupying a block with dimensions 25 by 10 by 20 m with the initial conditions similar to the previous section. The charge is positioned at a corner of the block. Both reflective and non-reflective boundary conditions are imposed on the walls of the block to model blast propagation in a street canyon under the assumption that the charge is positioned at the ground level at the centre line of the street.

The following MATLAB script simulates this problem on a uniform mesh. The pressure and temperature fields for increasing times are presented in Figures 6 and 7. It is clearly seen the formation of a secondary shock wave.

```
%% 3D blast propagation in a street canyon
project_name = 'blast3d';
disp(sprintf('Project name: %s',project_name));
gds = gas_dynamics_solver;
R = 287.0; Kelvin0 = 273.0;
T0 = Kelvin0+20.0; p0 = 1.01e5; r0 = p0/T0/R;
T1 = Kelvin0+2000.0; p1 = 11*p0; r1 = p1/T1/R;
CFL = 0.5; save_time = 0.005; kmax = 6;

%% Pre-process if necessary
k = 0;
if ~exist(sprintf('%s%d.mat',project_name,k),'file')
    % Generate mesh
    s.LX = 25.0; s.LY = 10.0; s.LZ = 20.0;
    s.NX = 100; s.NY = 40; s.NZ = 80;
    dx = s.LX/s.NX; dy = s.LY/s.NY; dz = s.LZ/s.NZ; s.dx = min([dx,dy,dz]);
    s.x = -dx/2:dx:s.LX+dx/2; s.y = -dy/2:dy:s.LY+dy/2; s.z = -dz/2:dz:s.LZ+dz/2;
    NC = (s.NX+2)*(s.NY+2)*(s.NZ+2);
    NF = (s.NX+1)*s.NY*s.NZ+s.NX*(s.NY+1)*s.NZ+s.NX*s.NY*(s.NZ+1);
    disp(sprintf('%d facets, %d cells',NF,NC-8));
    s.dv = (dx*dy*dz)*ones(NC,1); s.fc = zeros(NF,2); s.ds = zeros(NF,3);
    s.BE = zeros(s.NY*s.NZ,1); s.GE = zeros(s.NY*s.NZ,1); % east BC
    s.BW = zeros(s.NY*s.NZ,1); s.GW = zeros(s.NY*s.NZ,1); % west BC
    s.BS = zeros(s.NX*s.NZ,1); s.GS = zeros(s.NX*s.NZ,1); % south BC
    s.BN = zeros(s.NX*s.NZ,1); s.GN = zeros(s.NX*s.NZ,1); % north BC
    s.BB = zeros(s.NX*s.NY,1); s.GB = zeros(s.NX*s.NY,1); % bottom BC
    s.BT = zeros(s.NX*s.NY,1); s.GT = zeros(s.NX*s.NY,1); % top BC
    s.map = @(ix,iy,iz) sub2ind([s.NX+2,s.NY+2,s.NZ+2],ix,iy,iz);
    i = 0; % facet index
    for ix = 1:s.NX+1
        for iy = 2:s.NY+1
            for iz = 2:s.NZ+1
                i = i+1; s.ds(i,:) = [dy*dz,0,0];
                s.fc(i,1) = s.map(ix,iy,iz);
                s.fc(i,2) = s.map(ix+1,iy,iz);
            end
        end
    end
    for ix = 2:s.NX+1
        for iy = 1:s.NY+1
            for iz = 2:s.NZ+1
                i = i+1; s.ds(i,:) = [0,dx*dz,0];
                s.fc(i,1) = s.map(ix,iy,iz);
                s.fc(i,2) = s.map(ix,iy+1,iz);
            end
        end
    end
    for ix = 2:s.NX+1
        for iy = 2:s.NY+1
```

```
            for iz = 1:s.NZ+1
                i = i+1; s.ds(i,:) = [0,0,dx*dy];
                s.fc(i,1) = s.map(ix,iy,iz);
                s.fc(i,2) = s.map(ix,iy,iz+1);
            end
        end
    end
    for iy = 2:s.NY+1
        for iz = 2:s.NZ+1
            i = sub2ind([s.NY,s.NZ],iy-1,iz-1);
            s.BE(i) = s.map(2,iy,iz); s.GE(i) = s.map(1,iy,iz);
            s.BW(i) = s.map(s.NX+1,iy,iz); s.GW(i) = s.map(s.NX+2,iy,iz);
        end
    end
    for ix = 2:s.NX+1
        for iz = 2:s.NZ+1
            i = sub2ind([s.NX,s.NZ],ix-1,iz-1);
            s.BS(i) = s.map(ix,2,iz); s.GS(i) = s.map(ix,1,iz);
            s.BN(i) = s.map(ix,s.NY+1,iz); s.GN(i) = s.map(ix,s.NY+2,iz);
        end
    end
    for ix = 2:s.NX+1
        for iy = 2:s.NY+1
            i = sub2ind([s.NX,s.NY],ix-1,iy-1);
            s.BB(i) = s.map(ix,iy,2); s.GB(i) = s.map(ix,iy,1);
            s.BT(i) = s.map(ix,iy,s.NZ+1); s.GT(i) = s.map(ix,iy,s.NZ+2);
        end
    end
    % Initial conditions
    xa = 0.0; ya = 0.0; za = 0.0; a = 5.0; b = 100.0;
    density = @(x,y,z)...
        0.5*((r1+r0)+(r1-r0)*tanh(...
        b*(a-sqrt((x-xa)^2+(y-ya)^2+(z-za)^2))));
    pressure = @(x,y,z)...
        0.5*((p1+p0)+(p1-p0)*tanh(...
        b*(a-sqrt((x-xa)^2+(y-ya)^2+(z-za)^2))));
    s.r = zeros(NC,1); s.p = zeros(NC,1); s.v = zeros(NC,3);
    for ix = 1:s.NX+2
        for iy = 1:s.NY+2
            for iz = 1:s.NZ+2
                i = s.map(ix,iy,iz);
                s.r(i) = density(s.x(ix),s.y(iy),s.z(iz));
                s.p(i) = pressure(s.x(ix),s.y(iy),s.z(iz));
            end
        end
    end
    % Specify Riemann solver
    s.gamma = 1.4; rs = riemann_solver(s.gamma);
    s.state = rs.interface_state; s.t = 0.0; s = gds.initialise(s);
    eval(sprintf('save %s%d -struct s',project_name,k));
end

%% Find the latest restart file
while exist(sprintf('%s%d.mat',project_name,k),'file')
    k = k+1;
end
k = k-1;

%% Load start/restart file
s = load(sprintf('%s%d',project_name,k));
while k < kmax
    elapsed_time = cputime;
    k = k+1;
    tf = s.t + save_time;
    while s.t < tf
        s = gds.update_rates(s);
        dt = CFL*s.dx/max(s.ia+abs(s.ivn));
```

```
            dt = min(dt,tf-s.t); s.t = s.t+dt;
            disp(sprintf('t = %g',s.t));
            s.r = s.r+s.r_rate*dt;
            s.e = s.e+s.e_rate*dt;
            s.u = s.u+s.u_rate*dt;
            % reflective east BC (symmetry plane)
            s.r(s.GE) = s.r(s.BE); s.e(s.GE) = s.e(s.BE);
            s.u(s.GE,1) = -s.u(s.BE,1); s.u(s.GE,2) = s.u(s.BE,2);
            s.u(s.GE,3) = s.u(s.BE,3);
            % non-reflective west BC
            s.r(s.GW) = s.r(s.BW);  s.e(s.GW) = s.e(s.BW);
            s.u(s.GW,1) = s.u(s.BW,1); s.u(s.GW,2) = s.u(s.BW,2);
            s.u(s.GW,3) = s.u(s.BW,3);
            % reflective south BC (symmetry plane)
            s.r(s.GS) = s.r(s.BS); s.e(s.GS) = s.e(s.BS);
            s.u(s.GS,1) = s.u(s.BS,1); s.u(s.GS,2) = -s.u(s.BS,2);
            s.u(s.GS,3) = s.u(s.BS,3);
            % reflective north BC
            s.r(s.GN) = s.r(s.BN); s.e(s.GN) = s.e(s.BN);
            s.u(s.GN,1) = s.u(s.BN,1); s.u(s.GN,2) = -s.u(s.BN,2);
            s.u(s.GN,3) = s.u(s.BN,3);
            % reflective bottom BC
            s.r(s.GB) = s.r(s.BB); s.e(s.GB) = s.e(s.BB);
            s.u(s.GB,1) = s.u(s.BB,1); s.u(s.GB,2) = s.u(s.BB,2);
            s.u(s.GB,3) = -s.u(s.BB,3);
            % non-reflective top BC
            s.r(s.GT) = s.r(s.BT); s.e(s.GT) = s.e(s.BT);
            s.u(s.GT,1) = s.u(s.BT,1); s.u(s.GT,2) = s.u(s.BT,2);
            s.u(s.GT,3) = s.u(s.BT,3);
            % synchronise physical variables
            s = gds.update_states(s);
        end
        elapsed_time = cputime-elapsed_time;
        disp(sprintf('CPU time: %g secs / %g mins / %g hours',...
            elapsed_time,elapsed_time/60,elapsed_time/3600));
        eval(sprintf('save %s%d -struct s',project_name,k));
    end

    %% Post process
    disp('Generating PNG images...');
    set(clf,'Renderer','ZBuffer');
    k = 0;
    while exist(sprintf('%s%d.mat',project_name,k),'file')
        s = load(sprintf('%s%d',project_name,k));
        sx = s.x(s.NX+1); sy = [s.y(2),s.y(s.NY+1)]; sz = s.z(2);
        [x,y,z] = ndgrid(s.x,s.y,s.z); w = zeros(size(x));
        for ix = 2:s.NX+1
            for iy = 2:s.NY+1
                for iz = 2:s.NZ+1
                    i = s.map(ix,iy,iz);
                    w(ix,iy,iz) = s.p(i)/1000;
                end
            end
        end
        slice(...
            y(2:end-1,2:end-1,2:end-1),...
            x(2:end-1,2:end-1,2:end-1),...
            z(2:end-1,2:end-1,2:end-1),...
            w(2:end-1,2:end-1,2:end-1),...
            sy,sx,sz);
        daspect([1,1,1]); axis equal; colorbar;
        title(sprintf('Pressure (kPa) at time t = %g ms',1000*s.t));
        eval(sprintf('print -dpng %s%d_pressure',project_name,k));
        for ix = 2:s.NX+1
            for iy = 2:s.NY+1
                for iz = 2:s.NZ+1
                    i = s.map(ix,iy,iz);
                w(ix,iy,iz) = s.p(i)/s.r(i)/R;
            end
        end
    end
    slice(...
        y(2:end-1,2:end-1,2:end-1),...
        x(2:end-1,2:end-1,2:end-1),...
        z(2:end-1,2:end-1,2:end-1),...
        w(2:end-1,2:end-1,2:end-1),...
        sy,sx,sz);
    daspect([1,1,1]); axis equal; colorbar;
    title(sprintf('Temperature (K) at time t = %g ms',1000*s.t));
    eval(sprintf('print -dpng %s%d_temperature',project_name,k));
    k = k+1;
end
```

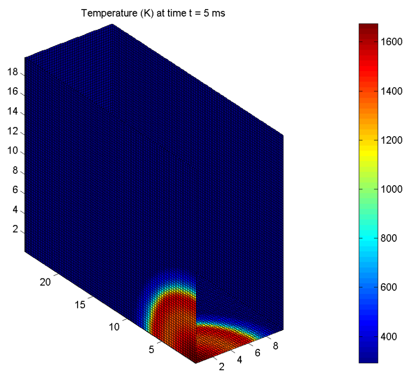


(i) $t = 5$ ms. (ii) $t = 10$ ms.

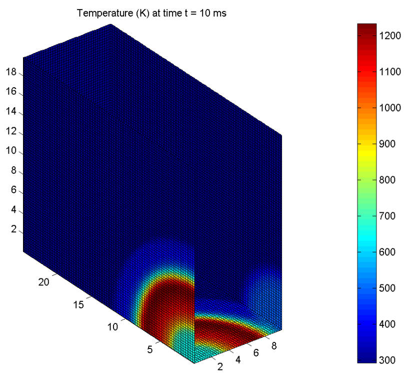
(iii) $t = 15$ ms. (iv) $t = 20$ ms.

(v) $t = 25$ ms. (vi) $t = 30$ ms.

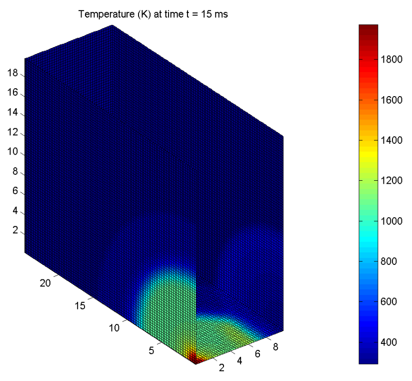
***Figure 6:*** *Three-dimensional blast propagation; pressure field slices.*



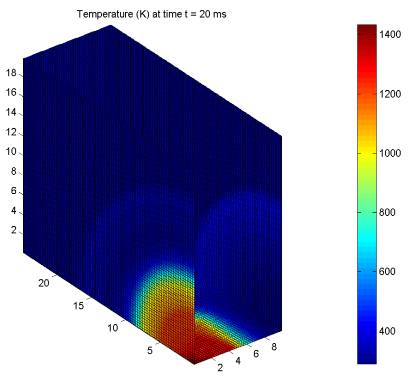
(i) $t = 5$ ms.



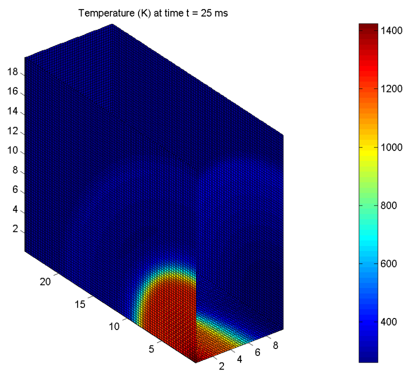
(ii) $t = 10$ ms.



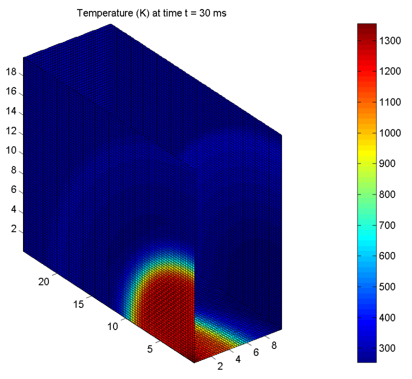
(iii) $t = 15$ ms.



(iv) $t = 20$ ms.



(v) $t = 25$ ms.



(vi) $t = 30$ ms.

***Figure 7:*** *Three-dimensional blast propagation; temperature field slices.*

# 7 Conclusion

The two reusable M-files constituting a Gas Dynamics Toolbox for MATLAB are sufficient for simulation of multi-dimensional problems for blast propagation. The comparison with the benchmark solution and the results of numerical simulation demonstrate that the code provides a reasonable solution which can be used for estimation of the peak pressure and impulse of shock waves.

It is worthwhile noting that the described numerical scheme is an implementation of the Finite Volume Method which, in turn, is a particular realisation of the Finite Element Method when both the shape and weighting functions coincide with the characteristic functions of cells $\omega_i$. The Finite Volume Method provides a conservative numerical scheme whose solution exhibits highly desirable properties of conservation of total mass, momentum and energy. Though the described first-order accurate Godunov-type solver is susceptible to numerical diffusion on coarse meshes, the total quantities such as the total load on part of a wall (e.g. windows and doors of a building) can be estimated quite reliably due to the conservativeness of the scheme.

# References

Brode, H. L. (1959) Blast wave from a spherical charge, *J. Phys. Fluids* **2**(2), 217–229.

Courant, R. & Friedrichs, K. O. (1948) *Supersonic Flow and Shock Waves*, Interscience, London.

Danaila, I., Joly, P., Kaber, S. M. & Postel, M. (2007) *Introduction to Scientific Computing: Twelve Computational Projects Solved with MATLAB*, Springer, New York.

Diestel, R. (2005) *Graph Theory*, 3rd edn, Springer, Berlin.

Gilat, A. (2008) *MATLAB®: An Introduction with Applications*, Wiley, New York.

Godunov, S. K. (1959) A difference scheme for numerical computation of discontinuous solutions of hydrodynamic equations, *Math. Sbornik* **47**(3), 271–306. In Russian.

Godunov, S. K. (1999) Reminiscences about difference schemes, *J. Comp. Phys.* **153**(1), 6–25.

Gottlieb, J. J. & Groth, C. P. T. (1988) Assessment of Riemann solvers for unsteady one-dimensional inviscid flows of perfect gases, *J. Comp. Phys.* **78**(2), 437–458.

Holt, M. (1977) *Numerical Methods in Fluid Dynamics*, Springer, Berlin.

Klee, H. (2007) *Simulation of Dynamic Systems with MATLAB® and Simulink®*, CRC Press, Boca Raton, FL.

Oran, E. S. & Boris, J. P. (1987) *Numerical Simulation of Reactive Flow*, Elsevier, New York.

Richtmyer, R. D. & Morton, K. W. (1967) *Difference Methods for Initial-Value Problems*, 2nd edn, Interscience Publishers, New York.

Roe, P. L. (1981) Approximate Riemann solvers, parameter vectors, and difference schemes, *J. Comp. Phys.* **43**(2), 357–372.

Sod, G. A. (1978) A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws, *J. Comp. Phys.* **27**(1), 1–31.

Toro, E. F. (1999) *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*, 2nd edn, Springer, Berlin.

van Leer, B. (1979) Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method, *J. Comp. Phys.* **32**(1), 101–136.

Page classification:

| | |
|---|---|
| **DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA** | 1. CAVEAT/PRIVACY MARKING |

| | |
|---|---|
| 2. TITLE<br>Solving multi-dimensional problems of gas dynamics using MATLAB® | 3. SECURITY CLASSIFICATION<br>Document (U)<br>Title (U)<br>Abstract (U) |
| 4. AUTHOR<br>L. K. Antanovskii | 5. CORPORATE AUTHOR<br>Defence Science and Technology Organisation<br>PO Box 1500<br>Edinburgh, South Australia 5111, Australia |

| | | | |
|---|---|---|---|
| 6a. DSTO NUMBER<br>DSTO–TR–2139 | 6b. AR NUMBER<br>014-204 | 6c. TYPE OF REPORT<br>Technical Report | 7. DOCUMENT DATE<br>June, 2008 |

| | | | | |
|---|---|---|---|---|
| 8. FILE NUMBER<br>2008/1023518/1 | 9. TASK NUMBER<br>NS 07/002 | 10. SPONSOR | 11. No OF PAGES<br>33 | 12. No OF REFS<br>16 |

| | |
|---|---|
| 13. URL OF ELECTRONIC VERSION<br>http://www.dsto.defence.gov.au/corporate/reports/DSTO–TR–2139.pdf | 14. RELEASE AUTHORITY<br>Chief, Weapons Systems Division |

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

*Approved For Public Release*

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DSTO RESEARCH LIBRARY THESAURUS

| | |
|---|---|
| Science | Simulation |
| Physics | Computational fluid dynamics |

19. ABSTRACT

This report describes an implementation of a Godunov-type solver for gas dynamics equations in MATLAB®. The main attention is paid to providing a generic code that can be easily adapted to particular problems in one, two or three dimensions. This is achieved by employing a cell connectivity matrix thus allowing one to use various structured and unstructured meshes without modification of the core solver. The code has been thoroughly tested for MATLAB Version 7.6 (Release 2008a).

Page classification: